NITTE EDUCATION TRUST | NITTE MEENAKSHI INSTITUTE OF TECHNOLOGY

An Autonomous Institution With A+ Grade By NAAC UGC | Approved By UGC / AICTE / Govt. of Karnataka | Bengaluru

DEPARTMENT OF ARTIFICIAL INTELLIGENCE AND MACHINE LEARNING

Natural Language Processing (22AM62)

FACULTY IN CHARGE: Dr. Zabiha Khan

UNIT-III

Learning to Classify Text: Supervised Classification, Further Examples of Supervised Classification, Evaluation, Decision Trees, Naive Bayes Classifiers, Maximum Entropy Classifiers, Modeling Linguistic Patterns

Extracting Information from Text: Information Extraction, Chunking, Developing and Evaluating Chunkers, Recursion in Linguistic Structure, Named Entity Recognition, Relation Extraction

Overview

Supervised Classification

Supervised classification is the task of assigning a label to an input based on patterns learned from labeled training data. In NLP, this often involves categorizing text documents, words, or sentences into predefined classes.

Components of supervised classification:

- Training data: Examples with known labels.
- Feature extractor: Converts input into a set of features (key-value pairs).
- Classifier: Learns how to assign labels based on features.

Example feature extractor for gender classification:

```
def gender_features(word):
  return {'last_letter': word[-1]}
```

Choosing the Right Features

Choosing effective features is critical for the performance of supervised classifiers. Good features are informative, non-redundant, and help the classifier generalize.

Common types of features used in NLP include:

- Lexical features: word forms, suffixes, prefixes
- Syntactic features: POS tags, chunk labels
- Contextual features: surrounding words, previous predictions
- Boolean features: presence or absence of specific words

Naive Bayes Classifier

The Naive Bayes classifier is a probabilistic model based on Bayes' Theorem with the 'naive' assumption that features are independent given the class.

Bayes' Theorem:

$P(\text{label} \mid \text{features}) \propto P(\text{label}) * \Pi\ P(\text{feature\_i} \mid \text{label})$

This classifier works surprisingly well for many NLP tasks such as spam filtering, sentiment analysis, and text categorization.

Advantages of Naive Bayes:

- Efficient training and classification
- Handles high-dimensional data
- Requires relatively small amounts of training data

Example in NLTK:

```
import nltk
from nltk.classify import apply_features

def gender_features(word):
   return {'last_letter': word[-1]}

labeled_names = [...]  # list of (name, gender) tuples
train_set = apply_features(gender_features, labeled_names)
classifier = nltk.NaiveBayesClassifier.train(train_set)
```

Evaluation

Evaluation metrics help assess the performance of classifiers. Common metrics include:

- Accuracy: Correct predictions / Total predictions
- Precision: TP / (TP + FP)
- Recall: TP / (TP + FN)
- F1 Score: Harmonic mean of precision and recall

- Confusion Matrix: Shows true positives, false positives, false negatives, and true negatives

Example:

```
nltk.classify.accuracy(classifier, test_set)
```

Decision Trees

Decision trees classify data by asking a series of questions about the features. Each internal node of the tree corresponds to a decision rule, and each leaf node corresponds to a class label.

Advantages:
- Easy to understand and interpret
- Handle both numerical and categorical data

Disadvantages:
- Prone to overfitting
- Can be unstable with small variations in data

Maximum Entropy Classifier (MaxEnt)

The Maximum Entropy classifier (also known as Logistic Regression) does not assume feature independence. Instead, it uses a model where each feature contributes a weight to the final classification decision.

The probability of a class is given by:
$P(c \mid x) = 1 / Z(x) * \exp(\sum_i w_i * f_i(c, x))$
Where:

- w_i: weight of feature i
- f_i: feature function
- Z(x): normalization term

Advantages of MaxEnt:

- Flexible and powerful
- Works well with overlapping and correlated features
- Often more accurate than Naive Bayes

Example in NLTK:

```
classifier = nltk.MaxentClassifier.train(train_set, algorithm='IIS', max_iter=10)
```

Modeling Linguistic Patterns

Supervised classifiers are used in many NLP tasks to model linguistic patterns. Examples include:

- Gender classification
- Part-of-speech tagging
- Named entity recognition
- Text classification

Example feature template for POS tagging:

```
def pos_features(sentence, i):
   features = {
      'word': sentence[i],
      'is_first': i == 0,
      'is_capitalized': sentence[i][0].upper() == sentence[i][0],
      'suffix-3': sentence[i][-3:],
      'prev_word': '' if i == 0 else sentence[i-1],
   }
   return features
```

Supervised Learning

Supervised learning with Support Vector Machines (SVM) is one approach to solving the WSD problem. Here's how it works:

1. **Data Collection**: To train an SVM for WSD, you need a labeled dataset where each word is tagged with its correct sense in various contexts. This dataset is typically created by human annotators who assign senses to words in sentences.

2. **Feature Extraction**: For each word in the dataset, you need to extract relevant features from its context. These features could include the words surrounding the target word,

part-of-speech tags, syntactic information, and more. These features serve as the input to the SVM.

3. **Training**: Once you have the labeled dataset and extracted features, you can train an SVM classifier. The goal is to teach the SVM to learn patterns in the features that are indicative of specific word senses.

4. **Testing/Predicting**: After training, you can use the SVM to predict the sense of an ambiguous word in a new, unseen sentence. The SVM considers the context features and assigns the word the most likely sense based on what it learned during training.

5. **Evaluation**: To assess the performance of your WSD system, you can use various evaluation metrics, such as accuracy, precision, recall, and F1-score. These metrics help you measure how well your SVM-based WSD system is performing in disambiguating word senses.

6. **Data Collection**: To train an SVM for WSD, you need a labeled dataset where each word is tagged with its correct sense in various contexts. This dataset is typically created by human annotators who assign senses to words in sentences.

7. **Feature Extraction**: For each word in the dataset, you need to extract relevant features from its context. These features could include the words surrounding the target word, part-of-speech tags, syntactic information, and more. These features serve as the input to the SVM.

8. **Training**: Once you have the labeled dataset and extracted features, you can train an SVM classifier. The goal is to teach the SVM to learn patterns in the features that are indicative of specific word senses.

9. **Testing/Predicting**: After training, you can use the SVM to predict the sense of an ambiguous word in a new, unseen sentence. The SVM considers the context features and assigns the word the most likely sense based on what it learned during training.

10. **Evaluation**: To assess the performance of your WSD system, you can use various evaluation metrics, such as accuracy, precision, recall, and F1-score. These metrics help you measure how well your SVM-based WSD system is performing in disambiguating word senses.

SVMs are popular for WSD because they are effective at handling high-dimensional feature spaces and can learn complex decision boundaries. However, the success of the SVM-based WSD system heavily depends on the quality of the labeled dataset and the choice of features used for training.

Unsupervised Learning

Unsupervised learning in Natural Language Processing (NLP) is a category of machine learning where the model is trained on unlabeled data without explicit supervision or predefined categories. It aims to discover patterns, structures, or representations within the data. One concept related to unsupervised learning in NLP is "Conceptual Density."

Semi Supervised Learning

Semi-supervised learning is a machine learning paradigm that combines both labeled and unlabeled data to improve model performance in the context of word-sense disambiguation

Decision tree

A decision tree is a simple model for supervised classification. It is used for classifying a single discrete target feature. Each internal node performs a Boolean test on an input feature. The edges are labeled with the values of that input feature. Each leaf node specifies a value for the target feature.

Classifying an example using a decision tree: Classifying an example using a decision tree is very intuitive. We traverse down the tree, evaluating each test and following the corresponding edge. When a leaf is reached, we return the classification on that leaf.

Example: Here is an example of using the decision tree.

Assume: I am 30 years old.

This is work related.

I am an accountant.

I am not trying to get red.

Following the tree, we answer no (not under 20 years old), no (not over 65 years old), yes (work related), no (not working in tech), and no (not trying to get red).

Problem: If we convert a decision tree to a program, what does it look like?
Solution: A decision tree corresponds to a program with a big nested if-then-else structure. First, we need to decide on an order of testing the input features. Next, given an order of testing the input features, we can build a decision tree by splitting the examples whenever we test an input feature.

Naïve Bayes Classifier:

To work out P([w1,w2,...,wn]|c)P([w1,w2,...,wn]|c), we make two assumptions on the data.

- First, just like in a bag-of-word representation, we assume word order doesn't matter.
- Second, we assume that each token appears independently of all other tokens.

This second assumption is where the term "naive" in Naive Bayes comes from. It is called naive because the assumption obviously isn't true in all languages. For example, if someone says the word "Merry" then proceeding word is much more likely to be "Christmas" than "cabbage". That is, "Merry" and "Christmas" are not independent words - they are in fact correlated. Hence, the naive assumption models the data sufficiently well.

Under these assumptions, we can express the likelihood as a product of conditional probabilities. In particular,

P([w1,w2,...,wn]|c)=P(w1|c)P(w2|c)...P(wn−1|c)P(wn|c)P([w1,w2,...,wn]|c)=P(w1|c)P(w2|c)...P(wn−1|c)P(wn|c).

And these are quantities we can calculate in the same way as we were doing before - for i=1,...,ni=1,...,n count. We can therefore update our model to be

^c=argmaxc∈CP(c)P(w1|c)P(w2|c)...P(wn−1|c)P(wn|c)(9)(9)c^=argmaxc∈CP(c)P(w1|c)P(w2|c)...P(wn−1|c)P(wn|c)

For very large corpuses, these probabilities will likely be **very small**. And multiplying lots of small numbers together can produce numbers too small for our computers to keep track of. Using properties of the logarithm, the **final Naive Bayes model** can be written as:

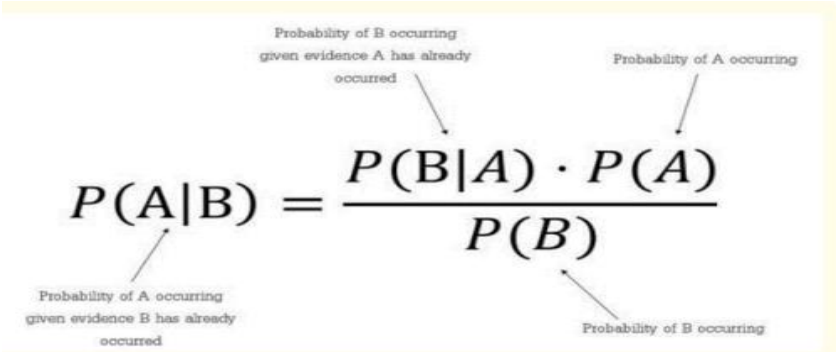


Figure 5: Probability of Classification using Naïve Bayes Theorem.

Naive Bayes is a **very fast** algorithm because it only requires **one forward pass** of the dataset.

Example: We are going to use the **20 news groups** sklearn dataset again and the test this time is given a news article; we need to classify which news group it belongs to.

```
# Fetch data
train_corpus = fetch_20newsgroups(subset='train', remove=('headers', 'footers', 'quotes'))
```

```
# Fetch data
train_corpus = fetch_20newsgroups(subset='train', remove=('headers', 'footers', 'quotes'))
test_corpus = fetch_20newsgroups(subset='test', remove=('headers', 'footers', 'quotes'))
```

```
# Print dataset sizes
print('Train size:', len(train_corpus.data))
print('Test size:', len(test_corpus.data))
```

Train size: 11314

Test size: 7532

And let's **preview the data**

```
#Labels
print('Labels:', train_corpus.target_names)
# Label encoding
print('\nLabel encoding:', train_corpus.target)
# Example article
print('\nExample article:', train_corpus.data[0])
```

Maximum Entropy Classifiers

A Maximum Entropy (MaxEnt) classifier is a probabilistic classifier that learns a model from training data by maximizing the entropy of the probability distribution, while adhering to constraints imposed by the data.

Maximum entropy (maxent) classifier has been a popular text classifier, by parameterizing the model to achieve maximum categorical entropy, with the constraint that the resulting probability on the training data with the model being equal to the real distribution.

The maxent classifier in *shorttext* is impleneted by *keras*. The optimization algorithm is defaulted to be the Adam optimizer, although other gradient-based or momentum-based optimizers can be used.

In Natural Language Processing (NLP), a maximum entropy (MaxEnt) classifier is a probabilistic classifier that uses the principle of maximum entropy to make predictions. It aims to find the most uniform

distribution (the one with the highest entropy) that satisfies the given constraints, which are often based on the observed data. This approach is particularly useful for text classification tasks like language detection, topic classification, and sentiment analysis.

- **Features:**

The ME model uses a set of features to describe the context of a word, such as:

- The word itself
- The preceding and following words
- The tag of the preceding word
- Word prefixes and suffixes

Advantages of ME models for POS tagging:

- **Flexibility:**

ME models can incorporate various features, including non-independent ones, making them adaptable to different languages and tagging tasks.

- **State-of-the-art Accuracy:**

ME models have been shown to achieve high accuracy in POS tagging, especially when combined with suitable features.

- **Handling Unknown Words:**

ME models can be used to predict the tags of unknown words by leveraging features that are similar to those of known words.

Example:

Imagine the sentence "The dog chased the ball." The ME model might consider features like:

- "The" is a determinative, "dog" is a noun, "chased" is a verb, and "ball" is a noun.
- The previous work "chased"
- The word's prefix and suffix (e.g., "chased" has the suffix "-ed")

Based on these features and the learned conditional probability model, the ME model would predict the most likely tag for each word in the sentence.

Modeling Linguistic Patterns

Linguistic patterns are recurring structures and rules in language, such as word order, grammar, and vocabulary usage. For example, the pattern of adding "-ing" to verbs to form gerunds is a linguistic pattern in English.

Applications of Modeling Linguistic Patterns:

- **Text Generation:**

Using the predicted probabilities to create new text, like in chatbots, creative writing tools, or even auto-complete features.

- **Machine Translation:**

Translating text from one language to another by understanding the linguistic patterns of both languages.

- **Speech Recognition:**

Converting spoken words into text by using language models to predict the most likely words based on the audio input.

- **Natural Language Understanding (NLU):**

Enabling computers to understand the meaning of human language by analyzing linguistic patterns.

- **Sentiment Analysis:**

Identifying the emotional tone or sentiment expressed in text by analyzing linguistic patterns related to emotion.

- **Medical Guideline Formalization:**

Using linguistic patterns to extract key information from medical guidelines and convert it into a formal representation.

- **Conceptual Modeling:**

Using linguistic patterns to model and transform information between different models.

Linguistic Patterns are grammatical rules that allow their users to write correctly in a common language. From a linguistic perspective, grammar is a collection of rules, but also a set of blueprints that guide speakers in producing more comprehensible and predictable sentences.

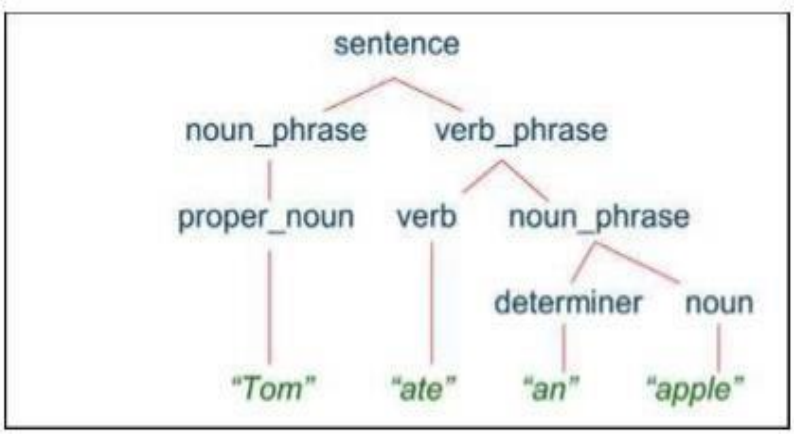
The parsing in NLP is the process of determining the syntactic structure of a text by analyzing its constituent words based on an underlying grammar.

Example Grammar:

```
sentence -> noun_phrase, verb_phrase
noun_phrase -> proper_noun
noun_phrase -> determiner, noun
verb_phrase -> verb, noun_phrase
proper_noun -> [Tom]
noun -> [apple]
verb -> [ate]
determiner -> [an]
```

Then, the outcome of the parsing process would be a parse tree, where sentence is the root, intermediate nodes such as noun_phrase, verb_phrase etc. have children - hence they are called non-terminals and finally, the leaves of the tree 'Tom', 'ate', 'an', 'apple' are called terminals.

Parse Tree:



- A sentence is parsed by relating each word to other words in the sentence which depend on it.
- The syntactic parsing of a sentence consists of finding the correct syntactic structure of that sentence in the given formalism/grammar.
- Dependency grammar (DG) and phrase structure grammar (PSG) are two such formalisms.
- PSG breaks sentence into constituents (phrases), which are then broken into smaller constituents.
- Describe phrase, clause structure Example: NP, PP, VP etc.,
- DG: syntactic structure consists of lexical items, linked by binary asymmetric relations called dependencies.
- These relations can also have labels.

- A treebank can be defined as a linguistically annotated corpus that includes some kind of syntactic analysis over and above part-of-speech tagging.

Information Extraction:

Information extraction (IE) is the process of automatically extracting structured information from unstructured or semi-structured sources, typically text-based documents like emails, reports, or web pages. This process transforms raw data into a format that is easier for computers to understand and use, allowing for tasks like database creation, knowledge base building, and data analysis.

Information Extraction (IE) in Natural Language Processing (NLP) is a crucial technology that aims to automatically extract structured information from unstructured text. This process involves identifying and pulling out specific pieces of data, such as names, dates, relationships, and more, to transform vast amounts of text into useful, organized information.

Importance of Information Extraction

1. **Enhancing Data Usability:** IE helps in converting unstructured text, which constitutes a significant portion of the data available, into structured formats that are easier to analyze and utilize.
2. **Automating Data Processing:** By automating the extraction process, IE reduces the need for manual data entry and analysis, saving time and resources.
3. **Supporting Decision-Making:** Extracted information can be used for decision-making in various areas such as healthcare, finance, and customer service, providing actionable insights from large datasets.

Key Components of Information Extraction

Named Entity Recognition (NER)

NER identifies and classifies entities within a text into predefined categories such as the names of persons, organizations, locations, dates, etc.

Relationship Extraction

This involves identifying and categorizing the relationships between entities within a text, helping to build a network of connections and insights.

Event Extraction

Event extraction identifies specific occurrences described in the text and their attributes, such as what happened, who was involved, and where and when it occurred.

Information Extraction Techniques in NLP

Here are the main techniques used in IE:

1. Named Entity Recognition (NER)

Definition: Identifying and classifying named entities (e.g., persons, organizations, locations, dates) in text.

Techniques:

- **Rule-based approaches:** Utilize predefined rules and patterns.
- **Statistical models:** Use probabilistic models like Hidden Markov Models (HMM) and Conditional Random Fields (CRF).
- **Deep learning:** Leverage neural networks such as BiLSTM-CRF and transformers like BERT.

2. Relation Extraction

Definition: Identifying and categorizing relationships between entities within a text.

Techniques:

- **Pattern-based:** Uses patterns and linguistic rules.
- **Supervised learning:** Employs labeled data to train classifiers.
- **Distant supervision:** Uses a large amount of noisy labeled data from knowledge bases.
- **Neural networks:** Utilizes CNNs, RNNs, and transformers for relation classification.

3. Event Extraction

Definition: Detecting events and their participants, attributes, and temporal information.

Techniques:

- **Template-based:** Matches text with pre-defined event templates.
- **Machine learning:** Uses classifiers and sequence labeling methods.
- **Deep learning:** Applies RNNs, CNNs, and attention mechanisms to capture event structures.

4. Coreference Resolution

Definition: Determining when different expressions in a text refer to the same entity.

Techniques:

- **Rule-based:** Employs heuristic rules.
- **Machine learning:** Trains classifiers using features like gender, number, and syntactic role.
- **Neural networks:** Uses deep learning models like BiLSTM and transformers for coreference chains.

5. Template Filling

Definition: Extracting specific pieces of information to populate predefined templates.

Techniques:

- **Rule-based:** Matches text to slots based on rules.
- **Machine learning:** Uses classifiers to fill template slots.
- **Hybrid methods:** Combine rules and machine learning for better accuracy.

6. Open Information Extraction (OpenIE)

Definition: Extracting tuples of arbitrary relations and arguments from text.

Techniques:

- **Pattern-based:** Utilizes linguistic patterns to identify relational triples.
- **Statistical:** Uses probabilistic models to determine the confidence of extracted relations.
- **Neural OpenIE:** Leverages deep learning models to improve the extraction process.

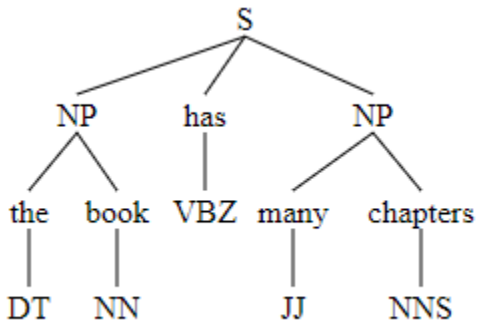
Chunking and the methods of Developing and Evaluating Chunkers

Chunks are made up of words and the kinds of words are defined using the part-of-speech tags. One can even define a pattern or words that can't be a part of chuck and such words are known as **chinks.**

```
from nltk.chunk import RegexpParser
# Introducing the Pattern
chunker = RegexpParser(r'''NP:{<DT><NN.*><.*>*<NN.*>} }<VB.*>{''')
chunker.parse([('the', 'DT'), ('book', 'NN'), ('has', 'VBZ'), ('many', 'JJ'), ('chapters', 'NNS')])
```

Output :

Tree('S', [Tree('NP', [('the', 'DT'), ('book', 'NN')]), ('has', 'VBZ'), Tree('NP', [('many', 'JJ'), ('chapters', 'NNS')])])



A ChunkRule class specifies what words or patterns to include and exclude in a chunk.

- The **ChunkedCorpusReader** class works similar to the TaggedCorpusReader for getting tagged tokens, plus it also provides three new methods for getting chunks.
- An instance of **nltk.tree.Tree** represents each chunk.
- Noun phrase trees look like Tree('NP', [...]) where as Sentence level trees look like Tree('S', [...]).
- A list of sentence trees, with each noun phrase as a subtree of the sentence is obtained in n chunked_sents()
- A list of noun phrase trees alongside tagged tokens of words that were not in a chunk is obtained in chunked_words().

In order to create an NP-chunker, we will first define a chunk grammar, consisting of rules that indicate how sentences should be chunked. The NP chunk should be formed whenever the chunker finds an optional determiner (DT) followed by any number of adjectives (JJ) and then a noun (NN). Using this grammar, we create a chunk parser , and test it on our example sentences graphically.

Example of a simple regular expression–based NP chunker.

```
>>> sentence = [("the", "DT"), ("little", "JJ"), ("yellow", "JJ"), , or display ... ("dog", "NN"), ("barked", "VBD"), ("at", "IN"),  ("the", "DT"), ("cat", "NN")]

>>> grammar = "NP: {<DT>?<JJ>*<NN>}"
>>> cp = nltk.RegexpParser(grammar)
>>> result = cp.parse(sentence)
>>> print result (S  (NP the/DT little/JJ yellow/JJ dog/NN)  barked/VBD  at/IN  (NP the/DT cat/NN))
>>> result.draw()
```

Chunking is a commonly used pre-processing step when ingesting documents for retrieval in the context of AI applications. Chunking serves to divide documents into units of information, with semantic content suitable for embedding-based retrieval and processing by an LLM.

For Example:

Chunking	Size	Overlap	Recall	Precision	Precision_Ω	IoU
Recursive	800 (~661)	400	85.4 ± 34.9	1.5 ± 1.3	6.7 ± 5.2	1.5 ± 1.3
TokenText	800	400	87.9 ± 31.7	1.4 ± 1.1	4.7 ± 3.1	1.4 ± 1.1
Recursive	400 (~312)	200	88.1 ± 31.6	3.3 ± 2.7	13.9 ± 10.4	3.3 ± 2.7
TokenText	400	200	88.6 ± 29.7	2.7 ± 2.2	8.4 ± 5.1	2.7 ± 2.2
Recursive	400 (~276)	0	89.5 ± 29.7	3.6 ± 3.2	17.7 ± 14.0	3.6 ± 3.2
TokenText	400	0	89.2 ± 29.2	2.7 ± 2.2	12.5 ± 8.1	2.7 ± 2.2
Recursive	200 (~137)	0	88.1 ± 30.1	7.0 ± 5.6	29.9 ± 18.4	6.9 ± 5.6
TokenText	200	0	87.0 ± 30.8	5.2 ± 4.1	21.0 ± 11.9	5.1 ± 4.1
Kamradt	N/A (~660)	0	83.6 ± 36.8	1.5 ± 1.6	7.4 ± 10.2	1.5 ± 1.6
⋆ KamradtMod	300 (~397)	0	87.1 ± 31.9	2.1 ± 2.0	10.5 ± 12.3	2.1 ± 2.0
⋆ Cluster	400 (~182)	0	91.3 ± 25.4	4.5 ± 3.4	20.7 ± 14.5	4.5 ± 3.4
⋆ Cluster	200 (~103)	0	87.3 ± 29.8	**8.0 ± 6.0**	**34.0 ± 19.7**	**8.0 ± 6.0**
⋆ LLM	N/A (~240)	0	**91.9 ± 26.5**	3.9 ± 3.2	19.9 ± 16.3	3.9 ± 3.2

Evaluation of various popular chunking strategies on our evaluation can have significant impact on retrieval performance, in terms of accuracy and efficiency. Size denotes chunk size in tokens, in brackets indicates mean chunk size where it may vary by chunking strategy. Overlap denotes the chunk overlap in tokens. Bold values highlight the best performance in each category.

RecursiveCharacterTextSplitter and **TokenTextSplitter** are some of the most popular chunking methods, and the default used by many RAG systems. These chunking methods are insensitive to the

semantic content of the corpus, relying instead on the position of character sequences to divide documents into chunks, up to a maximum specified length.

Chinking is the process of removing a sequence of tokens from a chunk. If the matching sequence of tokens spans an entire chunk, then the whole chunk is removed; if the sequence of tokens appears in the middle of the chunk, these tokens are removed, leaving two chunks where there was only one before. If the sequence is at the periphery of the chunk, these tokens are removed, and a smaller chunk remains.

Example 7-3. Simple chinker.

```
grammar = r"""
NP:
     {<.*>+}          # Chunk everything
 }<VBD|IN>+{      # Chink sequences of VBD and IN
""" sentence = [("the", "DT"), ("little", "JJ"), ("yellow", "JJ"),
 ("dog", "NN"), ("barked", "VBD"), ("at", "IN"),  ("the", "DT"), ("cat", "NN")]
cp = nltk.RegexpParser(grammar)
>>> print cp.parse(sentence)
(S
(NP the/DT little/JJ yellow/JJ dog/NN)
barked/VBD
at/IN
(NP the/DT cat/NN))
```

Recursion in Linguistic Structure

Building Nested Structure with Cascaded Chunkers

Trees consist of tagged tokens, optionally grouped under a chunk node such as NP. However, it is possible to build chunk structures of arbitrary depth, simply by creating multistage chunk grammar containing recursive rules.

Example has patterns for noun phrases, prepositional phrases, verb phrases, and sentences. This is a four-stage chunk grammar and can be used to create structures having a depth of at most four.

A chunker that handles NP, PP, VP, and S.

```
grammar = r"""
  NP: {<DT|JJ|NN.*>+}          # Chunk sequences of DT, JJ, NN
  PP: {<IN><NP>}               # Chunk prepositions followed by NP
  VP: {<VB.*><NP|PP|CLAUSE>+$} # Chunk verbs and their arguments
  CLAUSE: {<NP><VP>}           # Chunk NP, VP
  """
```

```
cp = nltk.RegexpParser(grammar)
sentence = [("Mary", "NN"), ("saw", "VBD"), ("the", "DT"), ("cat", "NN"),
   ("sit", "VB"), ("on", "IN"), ("the", "DT"), ("mat", "NN")]
>>> print cp.parse(sentence)
(S
  (NP Mary/NN)
  saw/VBD
  (CLAUSE
    (NP the/DT cat/NN)
    (VP sit/VB (PP on/IN (NP the/DT mat/NN)))))
```

Named Entity Recognition

Named entity recognition (NER) is a natural language processing (NLP) method that extracts information from text. NER involves detecting and categorizing important information in text known as *named entities*. Named entities refer to the key subjects of a piece of text, such as names, locations, companies, events and products, as well as themes, topics, times, monetary values and percentages.

NER is also referred to as *entity extraction*, *chunking* and *identification*. It's used in many fields in artificial intelligence (AI), including machine learning (ML), deep learning and neural networks. NER is a key component of NLP systems, such as chatbots, sentiment analysis tools and search engines. It's used in healthcare, finance, human resources (HR), customer support, higher education and social media analysis.

The goal of a **named entity recognition** (NER) system is to identify all textual mentions of the named entities. This can be broken down into two subtasks: identifying the boundaries of the NE and identifying its type. While named entity recognition is frequently a prelude to identifying relations in Information Extraction, it can also contribute to other tasks. For example, in Question Answering (QA), we try to improve the precision of Information Retrieval by recovering not whole pages, but just those parts which contain an answer to the user’s question.